by
Barry Simon

Lab Notes

Learning Your Way Around the Keyboard Under DOS, Part 1

■ DOS programmers and nonprogrammers alike will profit from knowing how this most basic of peripherals operates under DOS.

Even if you keep a mouse within reach, the keyboard is still your primary interface with the PC. But just how it tells programs what you want to do remains a mystery to most users, so in this Lab Notes I'll present an overview of how the keyboard works at the system level. Programmers will gain an understanding of what's going on under their high-level languages, and users will learn how to better manage their TSRs.

In this Lab Notes we'll be working entirely within the DOS world; both *Microsoft Windows* and OS/2 completely replace the default keyboard driver and so require a different treatment. In the DOS world, the briefest overview of keyboard operation can be given by saying that interrupt 9 processes keystrokes and places them in the keyboard buffer, while interrupt 16h is used by applications to obtain keystrokes from the keyboard buffer. As that terse summary suggests, however, to understand what the keyboard does will require some familiarity with hex numbering, 80*x*86 memory addressing, and interrupts, so it's a good idea to begin the discussion with these things.

WORKING IN HEX

Throughout this discussion we'll be using hexadecimal (base 16) numbers, known simply as hex. Where there's any chance of confusion, we place a lowercase *h* after such numbers as a reminder. Hex numbers need 16 symbols to represent them—six beyond the ten required for decimal notation—so following 0 through 9 come A, B, C, D, E, F. The number Eh is equal to 14 in decimal terms.

Just as the decimal number 123 means 1 times 100 (or 10^2) plus 2 times 10 (or 10^1) plus 3 times 1 (or 10^0), so the hex number 123h means 1 times 256 (16^2) plus 2 times 16 (16^1) plus 3 times 1 (16^0). Thus, 123h is 291 decimal, and, for a more complicated example at random, 13DEh becomes (1*4096 + 3*256 + 13*16 + 14), or 5,086 in decimal.

If you're unfamiliar with working in hexadecimal notation, it may be tempting to try to keep translating hex to and from decimal. In the situations in which hex numbers are used, however, you won't really gain anything by such translations. It doesn't matter that interrupt 16h is 22 decimal or that the start of the keyboard buffer (41Eh) is 1,054 in decimal. Hence, I recommend you regard hex numbers simply as themselves—numbers in base 16.

Even when you do have to do arithmetic, you'll eventually find it much easier to do it in hex, rather than by translating to decimal and back. Consider the following example:

```
 14FD h
 8FA4 h
------
 A4A1 h
```

You once had to learn that 13 + 4 was 17 by counting 14, 15, 16, 17. Likewise, at first you may have to learn that D + 4 = 11h by counting E, F, 10, 11. With practice, however, this process becomes second nature. And if it doesn't, most PC calculators have a hex mode, for the rare occasions that require hexadecimal arithmetic. Even DEBUG will give you the sum and difference, respectively, of two hex numbers if you enter

```
H nnnn mmmm
```

at the - prompt (where *nnnn* and *mmmm* are the hex numbers).

80x86 ADDRESSING SCHEME

A byte has 8 bits and so describes 2 to the power of 8, or 256 possibilities. Since one hex digit corresponds to 4 bits, it requires two hex digits to describe 1 byte. Bytes are thus written in the form *xx*, with each *x* representing a hex digit. Addresses are written in the form *xxxx:xxxx*, where the hex number before the colon is called the segment and the number after the colon is the offset. This curious way of writing addresses requires 4 bytes—2 for the segment and 2 for the offset—and is a consequence of the original design of the PC.

With 2 bytes—16 bits, or 4 hex digits—you could address 2^{16}, or 65,536 locations. The 8088/8086 CPUs used by the original PC were designed to access 1MB (that's 1,048,576 RAM locations)—a sizeable amount of memory at that time.

In absolute, or linear, address terms, this requires 20 bits (5 hex digits). Because the CPUs were internally designed to work with groups of 16 bits, however, the rather cumbersome segment:offset scheme—which uses 32 bits to represent an address that could have been expressed with 20 bits—was adopted.

Since there are 12 extra bits and 2^{12} is 4,096, each segment:offset address has 4,096 possible representations. This isn't

strictly true (there are fewer possible representations for addresses below 1,000h), but the principle is clear that the same absolute address in memory can be designated by many different segment:offset combinations. If you start with what is called a normalized address—one in which the segment is as large as possible and the offset as small as possible—you can derive the other 4,095 equivalent representations by repeatedly subtracting 1 from the segment and adding 16 (10h) to the offset. (If the segment is less than 4,096 (1000h), you can't actually do this 4,096 times, because you'll hit zero before you reach that number of iterations.)

To compute the corresponding absolute (linear) address from any of its segment: offset representations, you multiply the number to the left of the colon by 10h (that is to say, you shift it left by one digit) and then add the offset. Thus, the absolute address that corresponds to 1234:ABCD is

```
 12340
  ABCD
------
 1CF0D
```

Again, as with the hex/decimal conversion, it is usually not useful to translate from the 32-bit representation into absolute addresses except to determine whether two segment:offset addresses are actually names for the same location.

For example, address 0040:0010 is the same as 0000:0410. To determine this, you shift 0040 left one place (yielding 00400), add the 0010, and you get 00410, the same linear address. In general, you won't often need to create equivalent addresses. Rather, given two addresses, you'll want to know if they are equivalent. To do that, you just convert them both to linear and see if they're the same.

Given a normalized address, however, suppose you do want to find an equivalent address with a different segment. Let's assume the new segment is less than the starting segment by an amount that doesn't exceed 0FFFh. Take the difference between the two segment numbers, multiply it by 16 (10h), and add the result to the offset. That's the offset that goes with the new segment. So, for example, given 0040:0010, suppose you want the offset for segment 0010. The difference is 30h. Multiply by 10h (very easy, just shift it left one place) to get 300h. Hence 0010:0310 is an equivalent address.

WHERE YOUR KEYSTROKES GO

As I mentioned at the outset, processes connected with interrupt 9 put keystrokes into a memory buffer, and processes connected with interrupt 16 get them out. Now that we know where to find it, it's time to look at what goes on in the buffer, which, although the middle one, is the simplest element in the chain.

As shown in Figure 1, the first 400h (1,024 decimal) bytes in memory, from address 0000:0000 to 0000:03FF, are used by the CPU to store an interrupt vector table, which we'll discuss later on. Immediately above this (using equivalent addressing), the area from 0040:0000 to 0040:00FF is used by the BIOS—the low-level Basic Input/Output System—to store

data, including input from the keyboard. These 256 (decimal) memory bytes are known as the BIOS data area or the BIOS communication area. The last 16 bytes of the BIOS data area, called the Interprocess Communication Area, are intended to be used by programs to exchange data. In fact, this area is used only rarely, in part because there's no way to ensure that a third program won't overwrite what a first program places there for a second program.

To demonstrate just how keyboard information is stored in the BIOS data area, I've written a simple Turbo Pascal program, BIOSDATA.EXE, which you can download from PC MagNet. (Note: Instructions for downloading programs from PC MagNet, including those referenced in this Lab Notes, are printed in the article on this issue's free utility, WINWHERE .EXE. Commented source code is provided with the download.) BIOSDATA loads and continually rereads and displays the memory region from 0040:0000 to 0040:00FF, which gives you a real-time view of what is happening there. It also periodically reads the keyboard and exits when you hit Esc. The 256 bytes are displayed with the address at the extreme left, a hex display in the middle, and an ASCII display on the right, as seen in Figure 2. While it will be helpful to have the program actually on the screen as you read what follows, the discussion should be understandable without it.

BIOSDATA.EXE explains how keyboard information is stored in the BIOS data area.

Make sure CapsLock, NumLock, and ScrollLock are off and then load BIOSDATA. The first thing you'll notice is that two areas are obviously counting—the bytes at 0040:0040 and 0040:006C are changing rapidly with carryover to the neighboring bytes.

These have nothing to do with the keyboard, but, as you might guess, they are keeping time. Approximately 18.2 times a second, the computer's internal clock ticks, and special routines that change those bytes are run. The byte at address 0040:006C is counting the ticks since midnight; the other (at address 0040:0040) simply performs a countdown again and again, and is used with the mechanism that shuts off the disk drive motor.

Press and release the Right Shift key several times. You'll notice that the byte at address 0040:0017h changes back and forth from 0 to 1. Now try the same thing with Left Shift, then with the Ctrl key, and then with the Alt Key (use the left Ctrl and Alt if your machine has two of them). If you don't have BIOSDATA, what you would have seen is that they behaved just like Right Shift except that Left Shift showed a 2, Ctrl a 4, and Alt an 8. Now press Alt and Left Shift at the same time. You get 0A, which is decimal 10. That's exactly the sum of the Alt and Left Shift keys, that is, 8 + 2! The system stores the state of each shifting key independently in 4 of the bits of the byte at address 0040:0017.

If you now try turning CapsLock on and off, you'll notice two things. First, one of the bits in byte 0040:0018 tracks whether the CapsLock key is physically pressed down or not. Since this is a toggle rather than a shifting key, however, what's important is the CapsLock state. And whether CapsLock is on or off is kept in 0400:0017, the same byte that keeps track of the shifts. Thus all of the 8 bits in byte 0400:0017 are used: 4 for the four shifting keys and 4 for the CapsLock, NumLock, ScrollLock, and Insert toggles.

Four bits in the byte at 0040:0018 keep track of whether the toggle keys are depressed. For example, *XyWrite* can treat CapsLock as an extra shift by using this byte. The lower 4 bits on that byte go unused in the original PCs, XTs and early ATs. When the enhanced keyboard was introduced, however, these bits were put to use keeping track of the difference between the Left and Right Ctrl and Alt keys. If you have an enhanced keyboard and BIOSDATA, try experimenting with the left and right Alt keys while watching bytes 0040:0017 and 0040:0018.

Now, while watching the ASCII character display on the right of the screen, hit the 1 key on the top row repeatedly (or hold it down). You should see that the bytes in the addresses from 41Eh to 43Dh fill with alternating 1's and happy faces. This memory area is known as the keyboard buffer. If you look at BIOSDATA's hex display for this area, you'll see that 31h alternates with 02h. 31h is the ASCII code for the character 1; we'll see about the 02h's—the happy faces—shortly. Try hitting some other number or letter keys

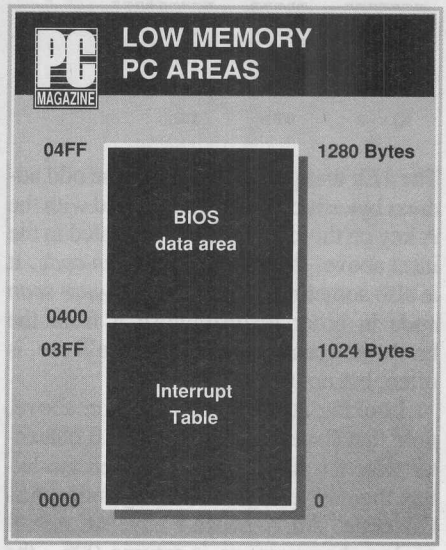


Figure 1: The low memory area from 40:0000 to 0040:00FF contains the keyboard buffer. This area is displayed in full by the BIOSDATA.EXE program.

while watching what happens in the ASCII representation of the keyboard buffer. You'll find that each keystroke sends a pair of bytes to that area, and that the one at the even address is shown on the ASCII display as the character you hit. I'll explain what happens to the odd bytes in a moment.

Next, try alternately hitting the top row Plus key and the gray Plus key. The ASCII part is, not surprisingly, a plus, mirrored by 2Bh's (the ASCII value of the plus sign) in the even addresses in the hex part of the display. But notice that the other half of what is placed in this area is different for the two keys. In the odd addresses, you'll see 0Dh's for the top row Plus key and 4Eh's for the gray Plus key. If you've ever wondered how programs can distinguish between the two plus keys, now you know.

We've determined so far that for each keystroke, 2 bytes are placed in the keyboard buffer, the 32-byte memory area from 0040:001E to 0040:003D. The even address byte gets an ASCII code. The odd address byte gets a code that can distinguish between similar keys. To confirm this, watch what happens when you push Ctrl-A, lowercase a, uppercase A, and Alt

Lab Notes

A. The results are shown below:

Key	ASCII	Scan Code
a	61h	1Eh
A	41h	1Eh
Ctrl-A	01h	1Eh
Alt-A	00h	1Eh

The 1Eh that goes into each of the odd address bytes is obviously connected with the A key on the keyboard. As indicated in the inset above, this is called the *scan code*. It is also sometimes called the *software scan code* in order to distinguish it from the hardware scan code which, as we'll see, is often, but not always, the same.

Looking at the middle column above, note that since Ctrl-A is an ASCII character (like the uppercase and lowercase letters themselves), it returns a genuine ASCII code. Alt-A doesn't have an ASCII analog, however, so it returns 00h—the NUL code. For keystrokes that produce a 00h for the ASCII code, the software scan code is sometimes called the extended ASCII code.

From this example, you might provisionally conclude that the scan code byte depends only on the key hit and that the ASCII code byte is used to distinguish among the shift states of that key. But if you now successively hit F1, Shift-F1, Ctrl-F1, and Alt-F1, you'll get the following results:

Key	ASCII	Scan Code
F1	00h	3Bh
Sft-F1	00h	54h
Ctrl-F1	00h	5Eh
Alt-F1	00h	68h

Since function keys do not have ASCII equivalents, their ASCII code should always be 00h, as it is. But, in order to distinguish between the shift states of the function keys, we must violate our new provisional rule that the code in the second half of the 2-byte combination depends only on the key hit, not on its shift state.

For the original PC keyboard the complete rule is this: In situations where the ASCII code distinguishes shift states (that is, where the ASCII codes are nonzero), the ASCII code byte depends only on the key hit, not on the shift state. In situations where the ASCII codes are all 0—which means the function keys and arrows—the scan code is the hardware scan code for the unshifted keys, a fixed but arbitrary convention for the shifted states.

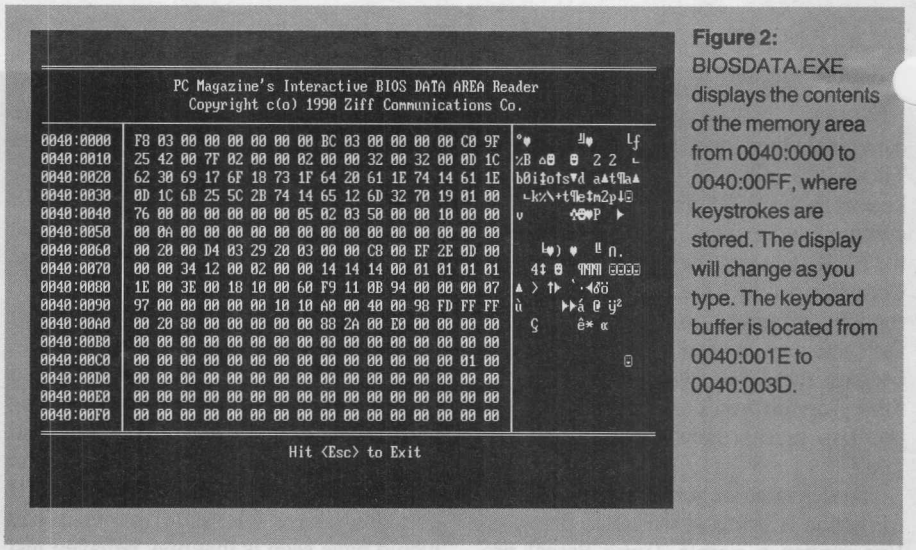


Figure 2: BIOSDATA.EXE displays the contents of the memory area from 0040:0000 to 0040:00FF, where keystrokes are stored. The display will change as you type. The keyboard buffer is located from 0040:001E to 0040:003D.

The enhanced keyboard adds a few extra complications. If you have one, try the two different Left Arrows; you'll find the following correspondence:

Key	ASCII	Scan Code
Left	00h	4Bh
Second Left	E0h	4Bh

Note that for the extra keys that were introduced with the enhanced keyboard, interrupt 9 places E0h in the ASCII part of the 2-byte sequence (since there is no key for which E0h is a valid ASCII value). While not many bother to do so, programs can

On the original PC keyboard, when the ASCII codes are nonzero, the software and hardware scan codes are the same.

GOING AROUND IN CIRCLES

While we've been watching the changes that take place in the keyboard buffer (the memory area from 0040:001E to 0040:003D), you may have noticed that as you hit various keys, the bytes at addresses 0040:001A and 0040:001C were also changing. One of these locations contain the address at which the next keystroke will be added (the tail). The other contains the address from which keystrokes will be removed (the head). BIOSDATA reads the keys as fast as they're entered, so the head keeps up with the tail.

The keyboard buffer is a queue, which is to say, keystrokes are added to the queue at the tail and removed at the head. The tail pointer is at 0040:001C, and 0040:001A is the head pointer. When these two pointers are equal, there are no keystrokes waiting in the buffer. If you've been typing various keystrokes into the buffer, why then does BIOSDATA always show the same values for 0040:001C and 0040:001A?

The reason is that the program is constantly on the lookout for an Esc, which means that you want to exit. It does this by checking for a keystroke before it updates the display. If there is a keystroke, it reads it from the keyboard buffer using INT 16h and exits if it is an Esc. As a result, it is taking keystrokes out of the buffer as fast as you can type them in. For this reason the head and tail of the buffer seem to be updated at the same time. Actually, on a slow machine (especially one with CGA where Turbo Pascal uses snow checking to slow the display), you will notice that 0040:001C gets ahead of 0040:001A.

They Left out Features....
We Left out the COMMA!!

The only thing missing...

is the comma in the price. If you look at the chart on the right you will see prices charged by our competition. All but one contain a comma. DesignCAD 3D sells for $399.00. Period. No Comma!

In order to draw the complex pictures shown below it is desirable to have the following 3D features:

- **Interactive design with 3D cursor**
- **Blending of surfaces**
- **Boolean operations such as add, subtract, and intersection**
- **Complex extrusions**
- **Cross sectioning**
- **Block scaling**
- **On screen shading**
- **Shaded output to printers and plotters**

All of these competitors left out one or more of these desirable features in their standard package. They didn't forget the most horrible feature - the comma.

DesignCAD 3D offers **ALL** the listed features plus many more!

If DesignCAD 3D has the power to create the 3D objects shown below, imagine how it could help with your design project!

DesignCAD 3D sells for $399. We left out the comma. We didn't think you would mind!

PC MAGAZINE SAYS...

DesignCAD 3D, the latest feature-packed, low-cost CADD package from American Small Business Computers, delivers more bang per buck than any of its low-cost competitors and threatens programs costing ten times as much. For a low-cost, self-contained 3D package... DesignCAD's range of features steals the show."

$399

AutoCAD rel. 10	$3,000.00	AutoCAD AEC $1,000.00 AutoShade $500.00
CADKEY 3.12	$ 3,195.00	Solids $995.00 IGES translator $1,995.00
DataCAD with DC Modeler	$ 3,990.00	DataCAD Velocity $2,000.00
DesignCAD 3D ver. 2.0	**$ 399.00**	**NO expensive options! IGES Free, Shading Free**
MaxxiCAD 1.02	$ 1,895.00	N/A
Mega Model	$ 995.00	MegaDraw $195, List $295, MegaShade $395
MicroStation PC 3.0	$ 3,300.00	Customer Support Libraries $1,000.00
ModelMate Plus 2.8	$ 1,495.00	N/A
VersaCAD Design 5.4	$ 2,995.00	N/A

Source: Byte Magazine

BYTE MAGAZINE SAYS...

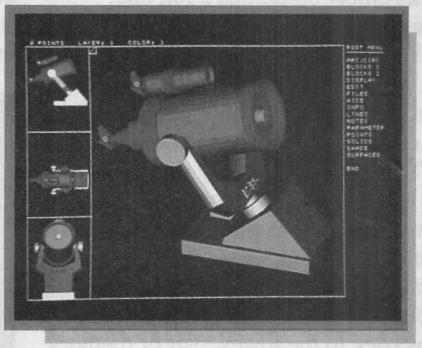
"At $399, DesignCAD 3D was the least expensive package we saw, yet it was one of the more powerful. ..Don't be fooled by the remarkably low price, this program can really perform!"

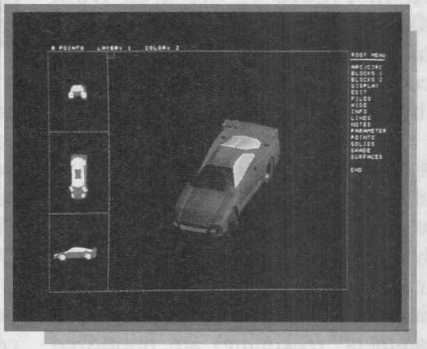
Complete 3-Dimensional design features make it easy for you to construct realistic 3-D models. With full solid-object modeling capabilities you can analyze your drawing to determine the volume, surface area or even center of gravity! DesignCAD 3-D even permits you to check for interference between objects! Aeronautical Engineers can now find the center of gravity for a new airplane design with a couple of keystrokes. The Architect can determine the surface area of a roof for decking in a matter of minutes. The Civil Engineer can calculate the volume of a lake or dam in seconds. The Mechanical Engineer will know for sure if certain parts fit together without interference. The uses for DesignCAD 3-D are only limited by YOUR imagination!

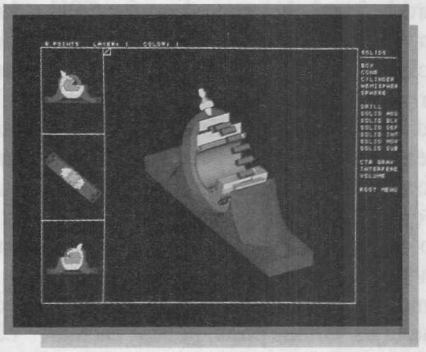
HOW DO I GET ONE?

DesignCAD 3-D and DesignCAD 2D are available from most retail computer stores, or you may order directly from us. If you have questions about which program to purchase please give us a call. All you need to run DesignCAD 3-D is an IBM PC or compatible computer with 640 K RAM memory and a hard disk. Both products support most graphics cards, printers, plotters and digitizers. Free Information and a demo disk are available by faxing (918) 825-6359 or telephoning:

1-(918) 825-4844







CIRCLE 475 ON READER SERVICE CARD

American Small Business Computers • 327 South Mill Street • Pryor, OK 74361 U.S.A.

The buffer is circular, so it loops around when the pointer reaches 0040:003D. If the last keystroke went into 0040:003C/0040:003D, the last byte in the buffer region, the next will go into 0040:001E/0040:001F, the first byte in the buffer.

Since it contains 32 bytes, you might think it would accommodate 16 keystrokes; for if the address of the head was fixed, there could be 16 possible places that the tail address can be. But since 0 strokes has to be a possibility, the 16 possibilities must be 0 . . . 15. Put differently, if one allowed 16 keystrokes, the tail would wrap around and catch up with the head, and there would be no way to distinguish an empty buffer and a full one!

This completes most of what I have to say about the BIOS data area. At this point, we haven't explained exactly how keystrokes get there, how programs get them out from there, and what programs like TSRs can do to modify the usual keyboard operations. We'll leave most of this for Part 2, but at this point we should take up the questions of what interrupts are in general and how they are generated by the keyboard hardware in particular. This will take us back to the lowest part of memory shown in Figure 1, the area from 0000:0000 to 0040:0000, and will also afford us a chance to explain the hardware scan codes that often end up in the odd-address bytes in the keyboard buffer.

INTERRUPTS AND SCAN CODES

The Intel 80*x*86 family is designed to use interrupts as the primary means by which both hardware and software communicate their needs to the microprocessor. Hardware interrupts ("true" interrupts by some programmers' reckoning) are prioritized by an interrupt controller chip on the motherboard before being passed on for CPU action. Thus, roughly 18.2 times a second, a clock chip stimulates the interrupt controller to cause the CPU to call interrupt 8 to update the time. And when you depress or release a key on the keyboard, the interrupt controller gets a signal and taps the CPU on the shoulder and tells it to call interrupt 9.

The alternative to an interrupt-driven system is to make the CPU continually run around to its various ports, polling each to see if one of them needs its attention. That's the only other way the CPU could learn that you've pressed a key. But periodically polling the keyboard port wastes a lot of CPU time—even when you're actively typing rather than looking at the screen. An interrupt system is like having students raise a hand if they have a question, as opposed to having the teacher call the class roll and ask each student if the point is clear.

When a program wants to read a keystroke from the buffer, it issues a software interrupt—specifically, interrupt 16. Applications also issue interrupts to call upon DOS services, such as reading from and writing to disk.

When an interrupt is called, either by hardware or software, a sequence of predetermined steps is executed. The routine that constitutes this sequence is called the interrupt handler. Whether this routine is actually located in the BIOS or elsewhere, a program must be able to access it by calling its interrupt number.

If a program wants to read a keystroke from the buffer, it issues software interrupt 16.

That's where the first 400h (1,024 decimal) bytes of the PC's RAM memory come in. This area of memory is called the interrupt vector table, and it can hold 100h or 256 addresses in segment:offset form. (You'll recall that memory addresses in the DOS world consist of a 2-byte segment and a 2-byte offset, thus requiring 4 bytes per address.) As part of the bootup process, the built-in ROM BIOS and the hidden system files place the addresses of known interrupt handlers—including the default routines for keyboard interrupts 9 and 16—in this table. Thus, for example, if interrupt 16h is called, execution immediately jumps to the 4-byte address starting at 0000:0058. If a program wants to enhance the services provided by the default routines, there's a way (that we'll describe in Part 2) for it to do this.

In addition to stimulating the interrupt controller to issue an interrupt 9 request, each press or release of one of the keys on the keyboard creates a unique hardware scan code. That code appears at the keyboard port (60h), where it is read by the interrupt 9 BIOS code and translated into the 2-byte sequence that goes into the keyboard buffer.

On the original PC, the hardware scan code for a key release was exactly 80h higher than for the code for the press of that same key. For example, the scan code for the a key is 1Eh on depress and 9Eh on release. Until it resets the keyboard, whenever the CPU looks at port 60h, this hardware scan code will be returned to it. On the enhanced keyboard, there is a difference for the new keys that issue multiple scan codes; for example, Up Arrow on the cursor pad is E0+4B on depress and E0+CB on release—there is nothing added to the E0.

The hardware scan codes are just a sequential numbering of the keys as they were physically laid out on the original PC keyboard, starting with Esc at 01h, the 1 key at 02h, and so on. From the original keyboard's point of view, the Shift keys were no different from the other keys. Since the left Shift key was adjacent to the \ key, their scan codes became 2Ah and 2Bh, respectively. As the keyboard changed, the keys migrated but kept their scan codes, with the extra keys getting new codes.

So that you can see for yourself what the hardware scan codes are for each of the keys on your machine, I've written a program, INT9.EXE. The program returns the set of hardware scan codes called by the interrupt 9 routines. INT9.EXE required some inline code in order to put its own handler inside the normal interrupt 9 loop. This inline code was taken from Turbo Power Software's Turbo Professional and is used here with permission. Because of the number of scan codes that some of the enhanced combinations produce, the display can be somewhat confusing, so I have placed a blank line after every keyboard-release scan code that occurs with no other strokes stacked. Depending on exactly when the interrupt 9s are processed, however, this will sometimes cause some extra blank lines in the sequence of eight codes that are produced when you depress and then release a cursor key with NumLock on.

How you get from the scan codes at port 60h to the 2-byte sequences held in the keyboard buffer will be our starting point in Part 2. ■

Barry Simon is a contributing editor of PC Magazine.

by
Barry Simon

Lab Notes

Learning Your Way Around the Keyboard Under DOS, Part 2

■ Here's how keystrokes are turned into the codes in the keyboard buffer and how applications, TSR utilities, and other keyboard programs use them.

As we saw at the end of Part 1, pressing a key on the keyboard deposits a hardware scan code at Port 60h and generates an interrupt 9 signal. The scan codes generated by the original PC keyboard, which were based simply on their physical location, are shown in Figure 1. As a comparison with Figure 2 shows, the introduction of the Enhanced keyboard made the situation somewhat more complicated by requiring additional scan codes for the newly added keys. With either keyboard, however, the BIOS routines invoked by interrupt 9 are responsible for reading the port and turning its codes into the 2-byte sequences (which we have called the ASCII and scan code bytes) that are deposited in the keyboard buffer, a low-memory area that begins at 0040:001E.

To process an ordinary key, for example, the a key, all that the interrupt 9 routines have to do is read the port, place an ASCII a code (61h) and the scan code (in this case the hardware and processed scan codes are the same: 1Eh) in the keyboard buffer area, and adjust the pointer to the tail of the buffer. The handler also needs to check that the buffer isn't full so that it can take appropriate action if it is. (The default is to issue a beep.)

Because the a key must produce different codes in the buffer to reflect the state of the shifting keys and CapsLock, even processing such a simple key becomes a more complex task. Among the steps that the interrupt 9 handler must take, between the striking of the key and the appearance of its 2-byte sequence in the keyboard buffer, are the following:

- If the key hit or released is a shifting key, adjust byte 417h, as explained in Part 1 (and demonstrated there with the BIOSDATA program).
- If the key hit or released is a lock key, adjust bytes 417h and 418h (see Part 1).
- If any one of four specific key combinations (or the Sys Req key on the AT) is pressed, invoke special processing routines directly. These key combinations are Ctrl-Alt-Del (to initial a reset); Shift-PrtSc (interrupt 5 is called); Ctrl-NumLock or Pause (go into a loop waiting for another key to be hit); or Ctrl-Break (but not Ctrl-C; see below).
- If Alt is pressed, check and specially process the keypad keys (for manually entering ASCII codes, as explained below).
- Determine the ASCII code (the even-address byte to be sent to the keyboard buffer) of the key combination, reflecting the status of Shifts, CapsLock, and NumLock. (Again, as noted in Part 1, on the Enhanced keyboard, E0h instead of 00h is used to distinguish scan codes that exist only on the Enhanced keyboard.)
- Determine the processed scan code (the odd-address byte to be put in the keyboard buffer). If the ASCII code is nonzero and not E0h, this is the hardware scan code, but for certain keys (such as the function keys when a shifting key is also pressed), it may be different.

With all this processing it's easy to see why the interrupt 9 code runs to about five pages in the AT BIOS listing. (The interrupt 16 code for getting keystrokes out of the buffer takes only about one page.) On the original PC, the typematic feature, which automatically repeats keystrokes if you hold a key down, was also handled by interrupt 9. On the AT and later machines this effect is produced by the keyboard itself. The change added flexibility: On an AT you can adjust typematic speed by sending commands to the keyboard's own on-board microprocessor.

On the XT, adjusting the typematic rates requires using a memory-resident program, which, as we shall see, invites the possibility of conflicts. Such speed-up programs simply "watch" the keyboard to see if you're holding down a key (that is, they've "seen" the keypress code for it, but not the release code) and after a certain time, they start stuffing extra copies of the held-down key into the buffer. The key itself doesn't repeat any more frequently, but the multiple copies put into the buffer make it appear to do so. This procedure can lead to the cursor overshooting its intended stopping point when the added keystrokes get processed after you have released the key. Enhanced typematic programs that will stop the cursor on a dime must operate by using interrupt 16 and can therefore work only in software.

SPECIAL KEY PROCESSING

Even though users think of Ctrl-C and Ctrl-Break as virtually identical keystrokes, they are processed very differently by the BIOS. To the BIOS, Ctrl-C is a key much like Ctrl-B: The pair of bytes 03h and 2Eh are placed into the buffer. On the other hand, when Ctrl-Break is hit, the BIOS calls interrupt 18, where special processing can be done by whoever is controlling that interrupt. When DOS loads, it points

interrupt 18 toward its own code and runs a routine to set a flag that it sees the next time it checks for Ctrl-C or Ctrl-Break. If, when it makes that check, DOS sees that Ctrl-C or Ctrl-Break has been hit, it calls interrupt 23. Although the end results of Ctrl-C and Ctrl-Break are the same for the default handling set up by DOS and BIOS, by placing different routines on interrupt 18 and 23, an application program can handle these keystrokes rather differently.

Via the Alt-keypad keys, you can manually enter any ASCII code, including those that can't be entered any other way, such as code 235, a Greek lowercase delta. You simply press down the Alt key, type 235 on the keypad, and when you release Alt, the keystroke is entered. What gets placed in the buffer is the ASCII code 235 (EBh), with a 0 in the scan code byte.

Enhanced keyboards must have special smarts to handle something like the separate keypad. Suppose you have NumLock on and press an arrow on the cursor pad. As you can see, by using the INT9 program presented in Part 1, the keyboard sends out a complicated set of codes. First, it sends the code that would indicate that you had pressed the Left Shift key, and then it sends the code for pressing the normal Left Arrow key. When you release the arrow, it sends the codes for lifting the arrow and then the code for lifting the shift. Thus it mimics hitting Shift-Arrow, which, when NumLock is on, will send an unshifted arrow! In fact, the situation is even more complicated: To indicate the special enhanced keys, the keyboard also sends E0h scan codes. Thus, when NumLock is on, the four codes E0h, 2Ah, E0h, 4Bh are all sent when you hit the cursor-pad Left Arrow key. As we shall see, under interrupt 16, service 10h, the ASCII code for such a combination is E0h rather than the 00h that the usual Left Arrow sends!

RETRIEVING KEYCODES

How do programs get keystrokes out of the buffer? They could just look in memory, take the keystroke, and adjust the buffer head pointer. But there is a better way. This involves requesting the information from the BIOS, taking advantage of the handling routines associated with interrupt 16. Going through the BIOS lowers the chance that memory-resident (TSR) programs will get in each other's way and also provides a good way for programs like macros to process and translate requests by applications.

When a program issues a software interrupt, the CPU saves the caller's return address on the stack and then jumps to the address stored in the interrupt table (which is stored in the lowest area of memory, as we saw in Part 1) for that interrupt. When the interrupt processing is complete, the IRET (Interrupt RETurn) command pops the return address off the stack and jumps back to it.

Most interrupts serve multiple purposes. The calling program tells the interrupt handler which service it wants by placing the "service number" in the AH register, one of the CPU's general storage areas. Often additional information is passed to the interrupt in other registers, and information is returned by the interrupt handler in those same registers.

Interrupt 16 is used to obtain information about keystrokes, and in the original PC, it offered three services. Service 02h returns the shift status, that is, the contents

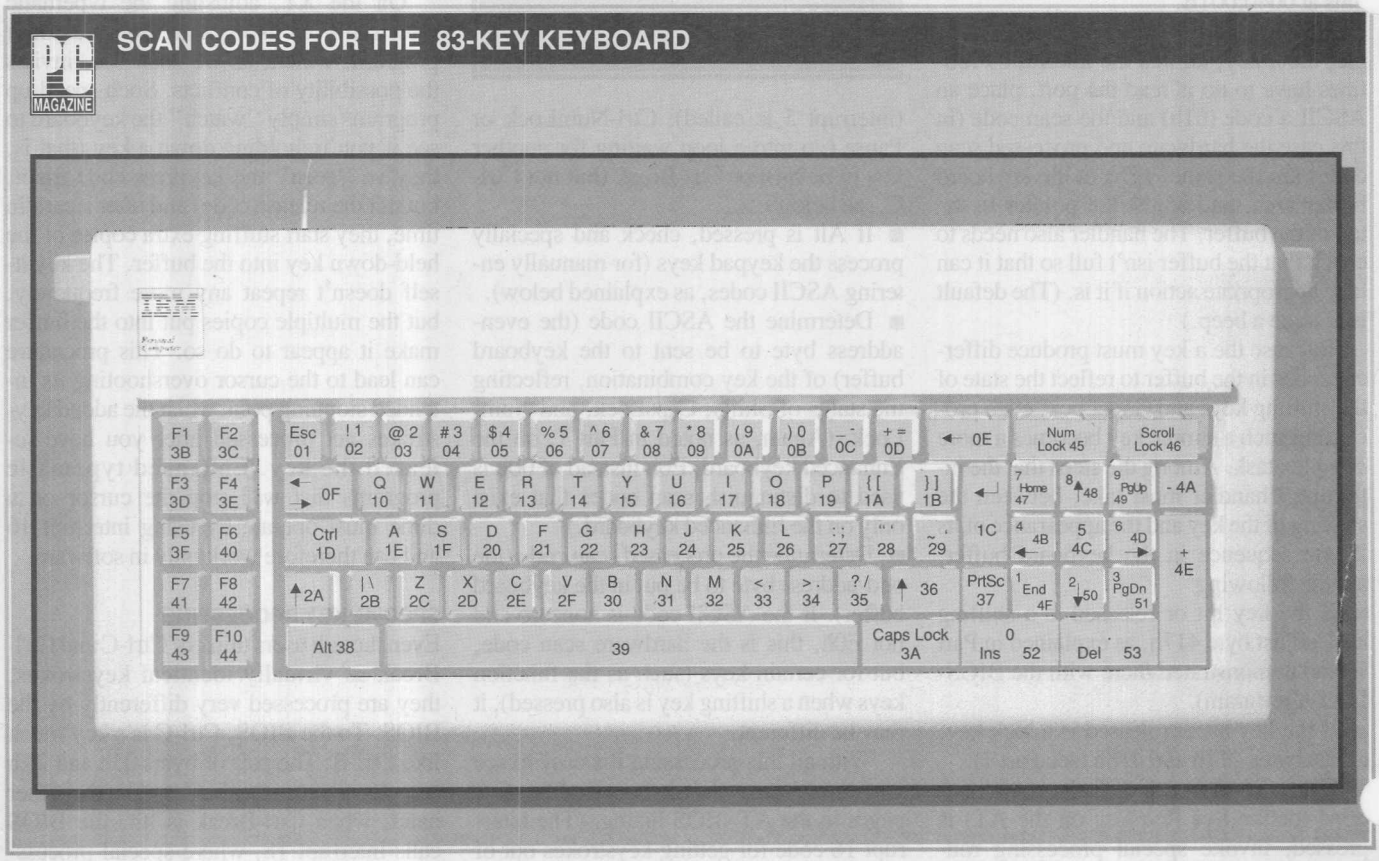


Figure 1: The original PC keyboard-generated hardware scan codes (shown in hex above) based on the spatial location of the keys on the keyboard.

Lab Notes

of byte 417h. This service is rarely used; it's easier for a program just to look at that byte for itself. (Since merely looking at a byte is nonintrusive, there is no reason to worry about conflicts.)

Service 01h asks whether there is a keystroke waiting. The answer is returned in one of the flags that are part of the register set. (Paradoxically, the flag is turned OFF if there is a keystroke waiting.) The keystroke, if present, is reported in the AH/AL registers, with the ASCII part in AL and the processed scan code in AH. Note, however, that when service 01h returns to the caller, it leaves any waiting keystroke in the buffer.

Service 00h is the main service for interrupt 16. It returns a keystroke in the AH/AL registers and removes the keystroke from the buffer. It doesn't actually delete the keystroke from memory but simply changes the address of the head of the buffer. With service 00h, if there is no keystroke waiting, control is not returned to the original program. Instead, the interrupt handler loops back, continually looking for a keystroke, and it returns only when a keystroke is ready.

Programs generally use one of two strategies to get keystrokes. Some just call service 00h and wait for the keystroke. Such programs will often spend most of their time looping in the interrupt handler. Multitasking systems like *DESQview* typically have special ways of handling such requests from applications so that background programs waiting for keyboard input do not get CPU time.

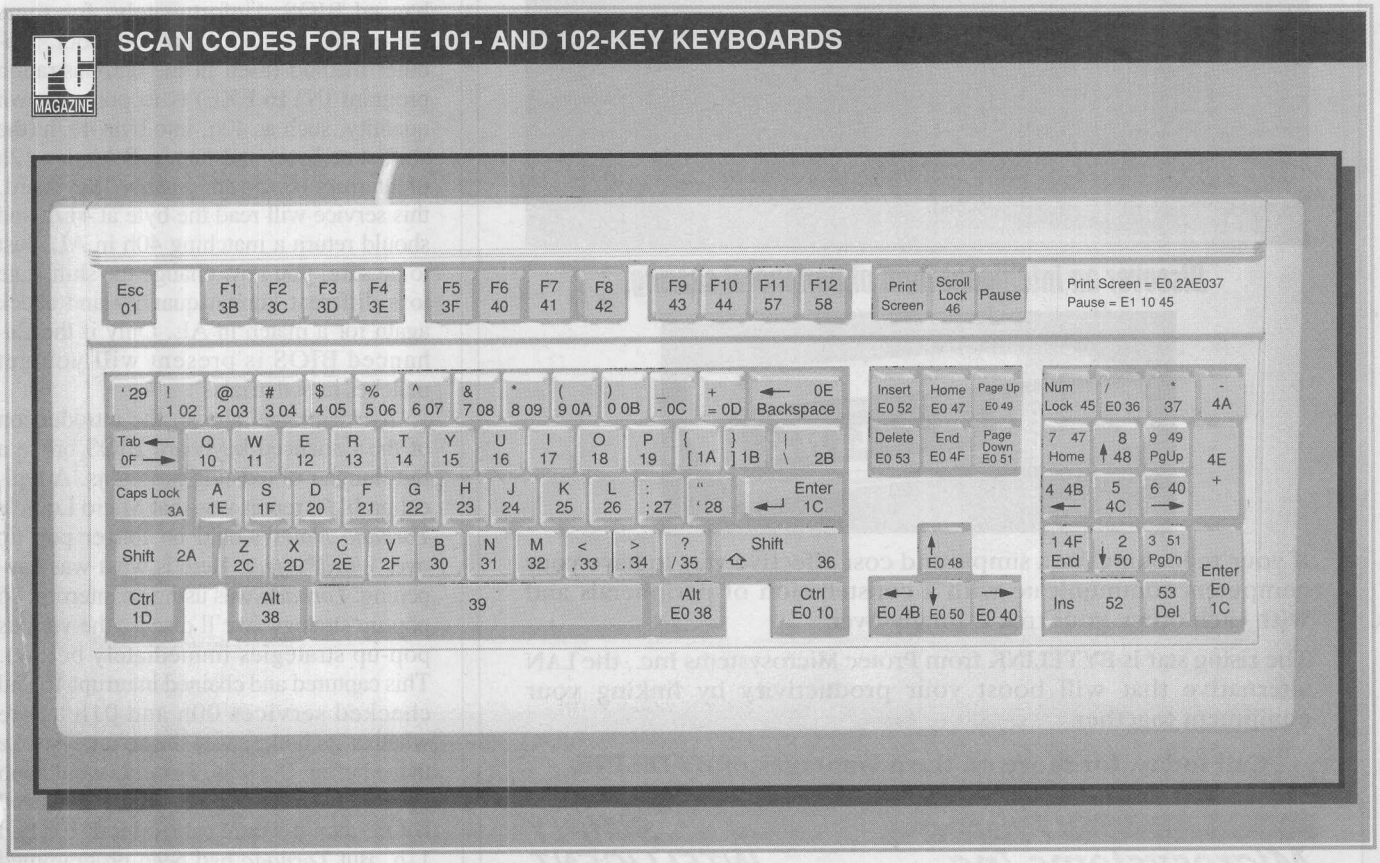
The other strategy is to use service 01h periodically, to check whether there is a keystroke waiting and to use service 00h only when service 01h sends back a positive report. Between requests to service 01h, the program can do housekeeping, such as updating an on-screen clock.

In the original PC, XT, and the first release of the AT, services 00h and 01h returned whatever bytes were in the buffer whether they were "legal" keystrokes or not. On a vanilla system, only legal keystrokes were in the buffer. Keyboard macro programs and programs like *DESQview*, however, would use illegal combinations, which they typically processed (usually by additions to the interrupt 16 routine) before the illegal codes could reach the underlying program.

With the advent of the Enhanced keyboard, initially on an AT but later on PS/2s, five new interrupt 16 services were provided. The most significant were services 10h and 11h, which are used to recognize new keys like F11 and F12.

Because the BIOS designers worried that a non-Enhanced keyboard program would become confused if it called service 00h and got the codes for F11—which it obviously couldn't know—Services 00h and 01h were rewritten to filter out keystrokes corresponding to combinations illegal under the old BIOS. Services 10h and 11h then allowed a program to read keys like F11. Service 10h is the analog of the old, unfiltered 00h, and 11h is the analog of 01h. The filtering broke a lot of TSRs, which had to rush to put out new versions in order to cope with the change in services 00h and 01h.

Service 12h, an analog of 02h, returns the bytes at both 417h and 418h. As we'll see, this service is often used for other than its nominal purpose of reading the shift status! Service 03h sets the typematic rate on



an AT keyboard, and 05 is used to stuff keystrokes into the buffer. Most programs that directly manipulated the keyboard buffer continued to do so, since they wanted to use the same routine for both pre- and post-Enhanced BIOS.

To illustrate how interrupt 16 works with both types of BIOS, I've written a simple program called INT16.EXE, which you can download from PC MagNet. (Note: Instructions for downloading programs, including all those referenced in this Lab Notes, are printed in the article on this issue's free utility, PCSPOOL.EXE. Commented Turbo Pascal source code is provided with the download.) Entered without its optional E parameter, INT16 calls interrupt 16, service 0 and returns the scan code and ASCII part in hex and decimal. It exits when you hit Esc. By entering INT16 E the program will instead call service 10h, though only if you have an Enhanced BIOS. (If you didn't have an Enhanced BIOS and service 10h were called, the program would never see an Esc, and you wouldn't be able to exit.) The program will inform you as to whether it's reading the normal or the Enhanced keyboard. If you have an Enhanced keyboard, you might try running this program both ways and see how F11 and Ctrl–Up Arrow are treated in each case.

The introduction of the Enhanced keyboard forced programs to make an important choice when asking for keystrokes. If they wanted to use F11, F12, the second keypad, and "new" keystrokes (such as Ctrl-Up, which the old BIOS did not support), the programs would have to call service 10h. If they did this on a machine with an old-style keyboard, they'd get nonsense and potentially could even hang.

Programs thus need a way to check for the presence of the Enhanced BIOS. A method sometimes used is to check the byte at 0040:0096. This is normally 00h on old-fashioned BIOS and not 00h on Enhanced BIOS. Unfortunately, for some clones this method is not foolproof. Another method (used in the demonstration program INT16.EXE) is to put a known quantity, such as 40h, into byte 417h (the shift-state byte) and then call service 12h of interrupt 16. On an Enhanced keyboard, this service will read the byte at 417h and should return a matching 40h in AL. Just to be sure, you now change the shift state to a different known quantity and check again for a match in AL. Only if the Enhanced BIOS is present will you get matches in both trials.

As mentioned above, the introduction of the Enhanced keyboard BIOS broke a fair number of existing programs. A typical problem report was that Micro Logic's *Tornado Notes* would no longer pop up over *WordPerfect*. Here is what was happening: *Tornado* was using an interrupt 16 pop-up strategy (we'll discuss the various pop-up strategies immediately below). This captured and chained interrupt 16 and checked services 00h and 01h to see whether its hotkey was the next keystroke in the buffer. If it was, *Tornado* would pop up. But if *WordPerfect* found an Enhanced BIOS, then it would call services 10h and 11h. But *Tornado* had been programmed to look only at services 00h and

Lab Notes

Logic has fixed the problem with the recent release of *Info Select,* a personal information manager.

TSRs AND KEYBOARD MACROS

One big advantage of using interrupts is that programs—memory-resident (TSR) pop-ups, keyboard enhancers, and the like—can interpose their own operations between the call to the interrupt and the execution of its handling routines. A program can do this simply by putting its own address at the point where the interrupt handler's address would ordinarily be found. The intervening program itself would then normally call the routine that was previously at that interrupt address, thus splicing its own operations into the instruction series. If several programs did this in succession, you would wind up with a chain of programs, each calling the next one. Not unnaturally, this procedure is called "chaining the interrupt."

The ever-popular TSR utilities all work by "grabbing an interrupt" and checking for a hotkey or shift state. This is one of the beauties of the interrupt structure. A program can leave its code resident and turn over the system to DOS, which allows programs to run. But the TSR, triggered by an external event or a service request from the underlying application, can periodically get control. Normally, all it does is see that it isn't wanted and passes control back. But when it sees its specific hotkey combination, it pops up and keeps control until you command it to pop down.

There are three common strategies for initiating the pop-up: interrupt 16, timer/shift, and interrupt 9. The interrupt 16 approach is rarely used; it depends on an underlying program requesting a keystroke. The way it works, however, is as follows: While the TSR is loading, it stores the old interrupt 16 address in its code segment and places an address of its own routine in the low-memory address called by interrupt 16. That is, it puts the address of its routine in the Interrupt Vector Table, at interrupt 16's "slot." Another TSR may load later and take interrupt 16, but as long as the new TSR properly chains interrupt 16, the first TSR will normally get control during each interrupt 16.

When the underlying program requests a keystroke with interrupt 16, the TSR gets the call. The TSR then calls the original interrupt 16 in such a way that it regains control before the underlying application does. It then checks the keystroke that interrupt 16 is returning. If the stroke is not its hotkey, the TSR passes control back to the underlying application. If it is the hotkey, the TSR pops up and interacts with the user.

Some TSRs will place their address in the interrupt table by directly manipulating memory. Interestingly enough, changing an interrupt vector is so basic to the PC's operation that programs can ask DOS to do it for them—and most TSRs do.

The second strategy involves chaining interrupt 8, the *timer interrupt,* which is triggered 18.2 times per second by the interrupt controller. During each interrupt 8, the TSR gets control briefly and looks at byte 417h to see if its shift hotkeys are pressed. If they aren't, the TSR passes control on. At first blush, it might seem to impose a terrible performance drag to have one or several TSRs getting control 18.2 times per second. But even on the slowest PC, there are over 50,000 CPU cycles between calls to interrupt 8 (there are almost ten times that on the fastest 33-MHz machines). A well-written assembly language routine will use only about 50 of these cycles to check that it isn't wanted, and even a compiled language won't use more than a few hundred.

TSR utilities all work by "grabbing an interrupt" and checking for a hotkey or shift state.

The third strategy is similar to that for interrupt 16, but it involves looking at interrupt 9. This means that the TSR can react immediately to the user's hitting a key instead of waiting for an underlying application to request a keystroke. In fact, so much work is done with interrupt 9 by a system like *DESQview* that modern TSRs do not pop up on interrupt 9. Rather, they set a flag (change a byte in their data area) and check for that flag using a separate interrupt 8 routine. The disadvantage of an interrupt 9 routine is that it can't be emulated by a keyboard macro program or a keyboard stuffer like *Keyfake*. But this is far outweighed by the fact that the interrupt 9 interrupt 8 routine works, and the others basically don't.

When a keyboard macro program is loaded, it chains into interrupt 16 so that when a program calls an interrupt 16, control is passed first to the macro program. The macro program, in turn, begins by calling the original interrupt 16 (whose address it has stored) in a way that control returns to the macro. The macro program can then check whether the keystroke in the buffer is one that it is supposed to translate. If it is, then it will return the translated keystroke to the underlying program. (Obviously the macro code will need to be complex in order to handle multikey translations, nested macros, and so on.)

Next, let's see how various keyboard enhancers work. How about a buffer expander to give you more than the 15 keystrokes of type-ahead? If you load the BIOSDATA demonstration (from Part 1) and look at the 4 bytes starting at address 480h, you should see

```
1E 00 3E 00
```

You'll recall that the keyboard buffer starts at 0040:001E, and the byte immediately after the buffer is 0400:003E. Words (16-bit quantities) are stored in memory with the lower half first, so 001E is stored as 1E first and then 00h. Hence, addresses 480h and 482h actually give the start and end of the buffer relative to 0040:0000.

In principle, a program could expand the buffer simply by changing those addresses. In fact, because not all machines support this scheme (some early Tandys didn't!), not all programs think to look there. Moreover, it isn't always possible to place an enlarged buffer in the 64K above 0400:0000 (which is where addresses 0400:*xxxx* lie), so no commercial program uses this expansion technique.

Still, simply as an experiment, I've written MOVEBUF.EXE, which is just like BIOSDATA but which starts by moving the pointers to B0 and F0. This gives you a 31-keystroke buffer. MOVEBUF also adjusts the bytes at 0040:001A and 0040:001C, which point to the head and tail of the buffer, so that they point instead to the new buffer region. Load MOVEBUF and try hitting keys. When you exit, it restores the buffer pointers. While I know of no BIOS that stores data in that area, there may be some, and in any event you should exercise care, since some BIOS may not act properly for other rea-

sons. But you'll probably find that this experimental double-size buffer works.

An alternative approach might involve rewriting interrupts 9 and 16 and moving the buffer elsewhere. Still, there very well may be conflicts. The buffer extender that comes with *SmartKey* does just this, and there are conflicts! Instead, a more satisfactory extender would modify the interrupt 9 Interrupt Service Routine (ISR). Unfortunately, explaining such a bullet-proof extender would take us beyond the scope of this Lab Notes.

MISSED OPPORTUNITIES

Only a few related keyboard issues remain to be discussed. First, you should know that there are some programs that use a DOS service to get keystrokes rather than calling interrupt 16. Never do this with your own programs, for the DOS service is brain dead! It calls interrupt 16 itself, but it normally throws away the processed scan code and returns only the ASCII code. If it always did this, it would throw away all keys whose ASCII code is 0! So in such cases the DOS service stores the scan code, and if you immediately follow the DOS call by a second one, it will return the scan code. This is more awkward than the interrupt 16 code, which almost all modern programs use. But to illustrate how far rampant nonsense can spread, although Turbo Pascal uses interrupt 16 rather than DOS, it mimics the DOS convention of requiring two calls to get function keys. I can only urge Turbo programmers to replace the standard readkey with one that uses interrupt 16 directly.

When the Enhanced BIOS and keyboard were introduced, extra code that calls interrupt 15, service 4Fh (with AL containing the scan code) was added to interrupt 9. Thus, a program can put itself in the interrupt 15 chain, get the keystroke, and prevent it from being processed further—all without needing to access the hardware or play other games. Many problems could be avoided if all TSRs used this method, but since not all machines support it, this useful code remains an untapped resource. Compatibility dictates that programs must continue to access interrupt 9 directly. Too bad!

Perhaps the biggest lost opportunity was in the original BIOS design. The 2 bytes stored in the buffer for each keystroke can describe 64K different combinations, and most of those just aren't used. For example, if the ASCII part is 2Bh (Plus), then only three of the 256 possible values for the other byte can occur: 0Dh (top row Plus key), 4Eh (gray Plus key), and 00h (entered via Alt-keypad as ASCII decimal 43). Imagine how much more useful it would be if the scan code and state of the shifts were stored with interrupt 16 doing the translation. There could then be an additional interrupt 16 service to give the raw data to a program that needed it.

But dreams of what might have been must give way to the realities that are, and I hope that this tour has given you a good idea of how your keyboard works and how programs, from normal applications to TSRs, interface with it. ■

Barry Simon is a contributing editor of PC Magazine.